Lista de Linguagens de Programação – 14

Nome: _____ Matrícula: _____

1. Existem três formas de alocação de memória: memória estática, memória de pilha e alocação em *heap*. Enquanto endereços estáticos são conhecidos em tempo de compilação, as outras formas de alocação usam endereços que serão conhecidos somente durante a execução do programa. Para cada variável do programa C abaixo, informe qual a forma de alocação de memória que será utilizada por ela:

```
int a = 7 ;
const int b = 9 ;
const int c = std::rand() ;

void foo() {
  int d = 9 ;
  int e = 9 ;
  const int f = std::rand() ;
  static int g = 7 ;
  static const int h = 9 ;
  static const int i = std::rand() ;
  int* j = (int*) malloc (sizeof(int)) ;
}
```

2. A classe `HeapManager`, disponível na página do curso (`HeapManager.py`), usa uma estratégia *first-fit* para encontrar o primeiro bloco de memória grande o suficiente para alocar uma requisição. Outro mecanismo bem conhecido, e simples, é chamado *best-fit*. Conforme você pode imaginar, esta estratégia consiste em percorrer a lista de blocos livres, em busca do pedaço de memória que seja o menor possível mas que seja grande o suficiente para comportar a área de memória requisitada. Se for encontrada uma área exatamente do tamanho da requisição, então pode-se interromper a busca, retornando a área encontrada. Do contrário, toda a lista deve ser percorrida, em busca do melhor pedaço de memória. A vantagem de *best-fit* é que esta estratégia não quebra áreas de memória muito grandes desnecessariamente. Se houver uma área de tamanho exato, *best-fit* a encontrará, não tendo, portanto, de quebrar nenhum bloco neste caso.

   Neste exercício você deve implementar uma nova versão da classe `HeapManager`, com esta política de alocação de memória. Comece com uma cópia de `HeapManager`, e então modifique o método `allocate` para implementar esta estratégia.

   Assim que você testar sua implementação, e verificar que ela funciona, ache uma sequência simples de requisições de memória para a qual a política *fist-fit* sucede e a política *best-fit* falha. *Dica*: existe uma sequência que começa deste jeito:

```
mm = HeapManager([0 for x in range(0, 11)]);
a = mm.allocate(4);
b = mm.allocate(1);
c = mm.allocate(3);
mm.deallocate(a);
mm.deallocate(c);
```

   Agora, você somente precisa estender esta sequência com três chamadas para `mm.allocate`, e você terá algo que terá sucesso com *first-fit* e falhará com best-fit.

3. Coleta de lixo é uma forma de gerenciamento automático de memória, e pode-se dizer que a vida seria bastante difícil para os programadores sem este tipo de ajuda. Difícil, mas não impossível, pois há linguagens, como C, que não possuem sistemas de coleta de lixo. Neste exercício você terá de fazer um pouquinho de pesquisa sobre coleta de lixo.

   (a) Qual linguagem introduziu mecanismos de coleta de lixo para o mundo? Quem foi o autor desta linguagem?

   (b) Conforme dito antes, algumas das linguagens mais populares não possuem nenhum mecanismo de coleta de lixo. C e C++ são bons exemplos. Descreva uma **desvantagem** de um serviço de coleta de lixo.

   (c) Escreva um parágrafo explicando cada um dos coletores de lixo a seguir: Marcação e varredura (*mark and sweep*) Cópia e coleta (*Copying*) e Contagem de referências (*Reference counting*)

4. Muitas linguagens de programação não possuem qualquer mecanismo de coleta automática de lixo. Um exemplo típico é C++. Ainda assim, é possível programar de forma mais segura via bibliotecas. Uma estratégia comumente adotada em C++ é baseada no uso de ponteiros desalocados automaticamente. Uma possível implementação deste tipo de ponteiro é dada logo abaixo:

```
template <class T> class auto_ptr {
  private: T* ptr;
  public:
    explicit auto_ptr(T* p = 0) : ptr(p) { }
    ~auto_ptr() { delete ptr; }
    T& operator*() { return *ptr; }
    T* operator->() { return ptr; }
};
```

   (a) A classe `auto_ptr` utiliza pelo menos dois tipos diferentes de polimorfismo. Que tipos de polimorfismos são estes?

   (b) A função abaixo contém um problema de mem?ória ou não? Em caso afirmativo, explique que falha é esta. Utilize a ferramenta `valgrind` para analisar este programa, por exemplo, tentando o comando `valgrind -v ./a.out`. Considere que uma falha de memória leva `valgrind` a fornecer algum aviso. Caso o erro não exista, justifique a sua resposta:

```
void foo0() {
  auto_ptr<std::string> p(new std::string("I did one P.O.F!\n"));
  p->print();
}
```

   (c) Novamente: problema de memória ou não? Em caso afirmativo, explique que falha é esta. Em caso negativo, justifique. Note que exceções, neste caso, funcionam como em Java ou Python:

```
void foo1() {
  try {
    auto_ptr<std::string> p(new std::string("Oi!\n"));
    throw 20;
  } catch (int e) { std::cout << "Oops: " << e << "\n"; }
}
```

(d) última pergunta: problema de mem?ória ou n?ão? Em caso afirmativo, explique que falha é esta. Em caso negativo, justifique a sua resposta:

```
void foo2() {
  try {
    std::string* p = new std::string("Oi!\n");
    throw 20;
    delete p;
  } catch (int e) { std::cout << "Oops: " << e << "\n"; }
}
```

5. Existem muitos mecanismos de coleta de lixo diferentes; alguns destes mecanismos servem domínios bem específicos de aplicações. Abaixo são dados dois diferentes cenários. Para cada um, descreva um mecanismo de coleta de lixo que seria interessante para ele. Neste curso falamos dos seguintes algoritmos: contagem de referências, marcação e varredura, cópia e coleta. Além destes, você pode pensar em outros, mais adequados à aplicação em questão.

   (a) Um sistema de tempo real, para controlar o braço de um robô que opera em uma linha de montagem. O principal requisito deste sistema é que ele deve responder a eventos em um certo período de tempo. Em hipótese alguma o sistema deve demorar mais que esta quantidade de tempo para produzir uma resposta.

   (b) Um servidor web. O servidor recebe milhares de requisições, todas elas independentes umas das outras. Uma característica interessante deste sistema é que cada requisição tem um período de vida curto, e leva à criação de uma quantidade pequena, e muitas vezes previsível de dados.